PLD Design Methodology

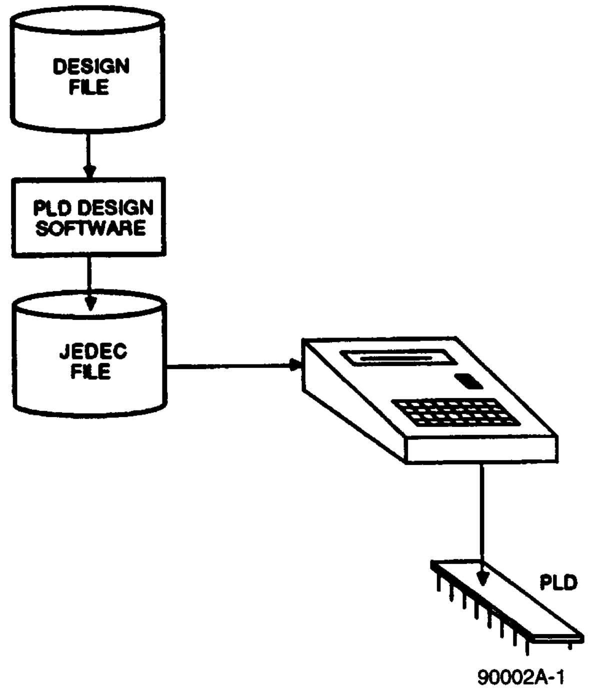
Programmable logic devices (PLDs) are used in digital systems design for implementing a wide variety of logic functions. These logic functions range from simple random logic replacement to complex control sequencers. Programmable logic devices offer the multiple advantages of low cost, high integration, ease of use, and easier design debugging capability not available in other systems design options. In the following discussion we will detail the PLD design process.

Most PLDs have an AND-OR array structure with programmable connections in either or both of the arrays. A programmable array implies that the connections can be programmed by the user. The popular PAL (Programmable Array Logic) devices have a programmable AND array and a fixed OR array. PAL devices are used for a wide variety of combinatorial and registered logic functions. In this discussion we will also examine the various design constraints to be considered when selecting the correct architecture for a given application.

All digital logic can be efficiently reduced to two fundamental gates, AND and OR, provided both true and complement versions of all input signals are available. Such logic is generally built around what is known as the sum-of-products (AND-OR) form. Programmable logic devices are ideal for implementing such two-stage logic in the AND and OR arrays.

Various process technologies offer many design options for PLDs. The connections in the programmable arrays can be fuse-based, commonly used in both ECL and TTL bipolar technologies, E/EEPROM cell based in UV-EPROM and EEPROM CMOS technologies, and RAM cell-based in CMOS RAM technology. The selection of technology is mostly dependent upon the system speed and power constraints. Most design engineers are familiar with these constraints, which not only dictate the technology of PLDs but also all of the other logic used in a system.

Designing with PLDs involves the use of design software and a device programmer (Figure 1). The design software eliminates the need to identify every connection to be programmed for implementing the desired sum-of-products logic. The design process begins with the creation of a design file which specifies the desired function. The function is typically represented by its sum-of-products form and can be derived directly from the timing diagram and/or truth tables. Occasionally Karnaugh maps and state diagrams are also used. The design file is then assembled to produce the "JEDEC" file. The JEDEC file gets its name from the fact that it is an approved JEDEC standard for specifying the state of every connection on the device. Simulation can then be performed. If the design is correct, the JEDEC file is downloaded into a device programmer for programming the connections on the device. The device can then be plugged into the PC board where it will function. The entire procedure can often be performed with the designer never having to leave the desk. Most programmers interface to personal computers, so that the design file can be edited, assembled, simulated, and downloaded, and the device programmed, all in one place.



90002A-1

Figure 1. PLDs are Designed Using Software and a Device Programmer

The first stage in a PLD design process (Figure 2) is the conceptualization of a design problem; the second is the selection of the correct device; the third is the implementation of the design, which also includes simulating the design with test vectors; and finally, the

Publication# 90002 Rev. A Amendment/0
Issue Date: June 1993

actual programming and testing on a system board. We will take a simple design example and go through the various stages of this design process.

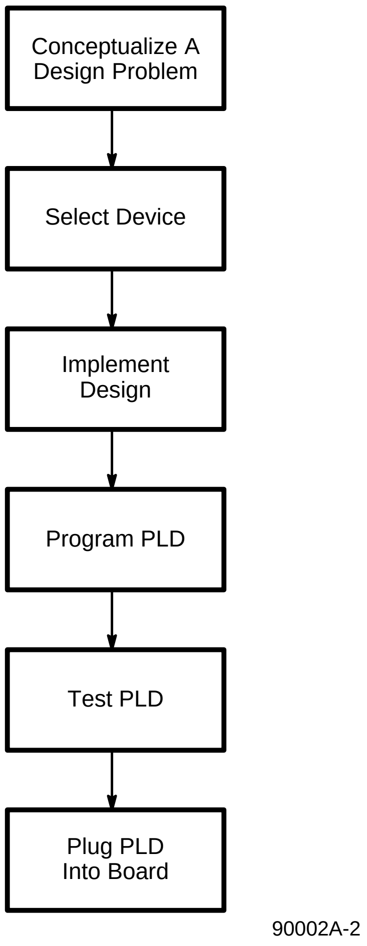


Figure 2. Programmable Logic Device Design Process

Conceptualizing a Design

The first step in the PLD design process is also required for any SSI/MSI design. An advantage of PLDs is that at this stage the designer needs to be concerned only with the required logic function. With SSI or MSI, various device logic limitations must be accounted for before the design can be started. Clearly a designer needs to develop a brief and complete functional description, based upon the system design requirements.

We will take the example of a simple address decoder circuit required for a 68000 microprocessor. The microprocessor has 24 address lines along with separate read and write signals. It requires some ROM to store the boot-up code as well as some RAM for storing and executing programs. The purpose of the address decoder circuitry is to select one of the memory addresses at a time. The RAMs and ROMs are assigned addresses on the 68000 microprocessor address space. The Address decoder circuit has to select one of the RAMs or ROMs for a specific range of addresses, called the address space. This selection is accomplished by asserting the specific chip-select signal for the RAM or ROM when the microprocessor accesses one of the addresses in the address space. There is additional circuitry in a typical microprocessor system for addressing I/O devices (such as disk controllers). These devices also require that chip-select signals be asserted when the microprocessor addresses them. Figure 3 shows an example address map for a 68000 microprocessor.

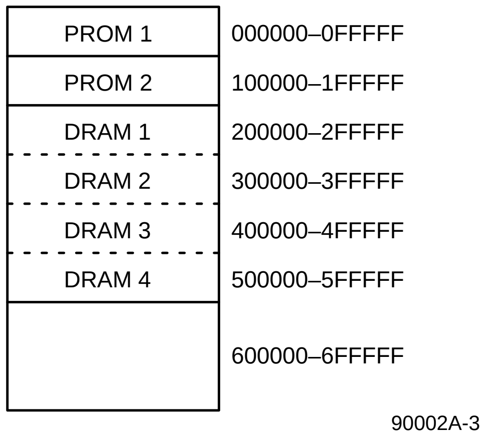


Figure 3. Memory Address Map

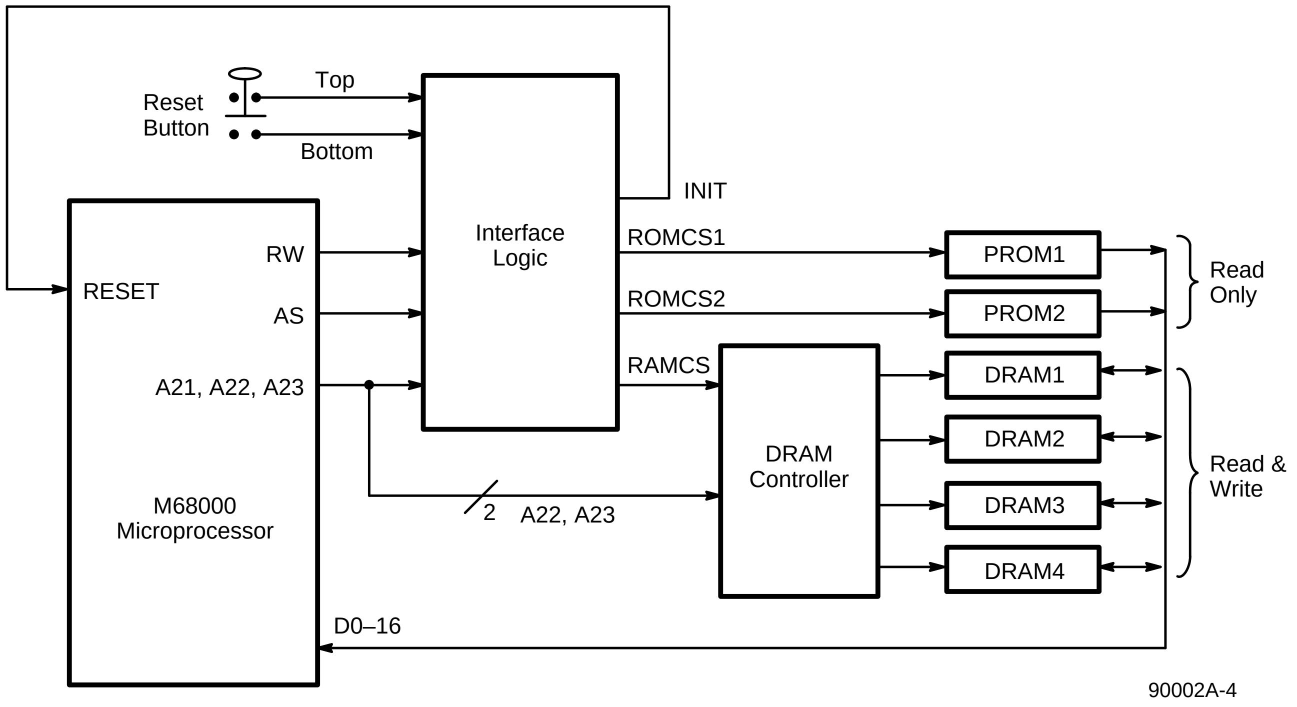


Figure 4. Microprocessor to Memory Interface

Figure 4 show the circuit diagram. The address signals from the 68000 microprocessor are inputs to the interface logic block. The outputs generated are ROMCS1, ROMCS2 and RAMCS. The generation of signals for selecting device I/Os is similar and is not shown here for the sake of simplicity. Other system inputs to the interface are the address strobe signal generated by the 68000 microprocessor as well as the read/write signal. The truth table for generating the outputs is shown in Table 1. This truth table is derived from the memory address map and the functional description of the design.

Table 1. Truth Table for Chip-Select Signals

Addresses Hex	Size	A23	A22	A21	Signal
000000–0FFFFF	1 MB	0	0	0	$\overline{\text{ROMCS1}}$
100000–1FFFFF	1 MB	0	0	1	$\overline{\text{ROMCS2}}$
200000–2FFFFF	1 MB	0	1	0	$\overline{\text{RAMCS}}$
300000–3FFFFF	1 MB	0	1	1	$\overline{\text{RAMCS}}$
400000–4FFFFF	1 MB	1	0	0	$\overline{\text{RAMCS}}$
500000–5FFFFF	1 MB	1	0	1	$\overline{\text{RAMCS}}$

Device Selection Considerations

The first task for the designer is to identify the design problem and classify it as a combinatorial function or a registered function, depending upon whether or not registers are required. In most cases, this decision depends upon the functional nature of the problem. Sometimes timing and logic considerations can also dictate the use of registers; this will be discussed later. Registers are usually not required for such simple combinatorial functions such as encoders, decoders, multiplexers, demultiplexers, adders, and comparators. However, registers are required for functions such as counters, timers, control signal generation, and state machines. No registers are required for this simple address decoding example.

The best choice for our combinatorial design would be a PAL device. The task now is to select a PAL device for implementing the desired function. General device selection considerations are listed below. These items are applicable to most designs.

- Number of input pins
- Number of output pins
- Number of I/O pins
- Device speed
- Device power requirements
- Number of registers (if any)
- Number of product terms
- Output polarity control

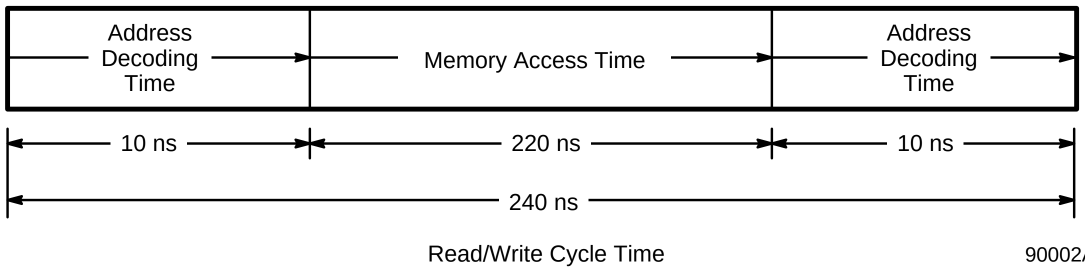


Figure 5. System Timing Requirements

The first resource that must be provided in a PLD is the number of pins needed for the basic logic function. This consists of the number of input and output pins. Many PLDs have internal feedback, which allows the generated output signal to be reused as an input. The same feedback also allows the pin to be used as a dedicated input, if required. This is especially useful for fitting various designs with different input/output requirements on the same device. The I/O pin capability of certain PLDs can also be very useful for certain bus applications.

The task is as simple as counting the number of input, output and I/O pins required by the design and picking a PLD which has the requisite number of pins.

The next selection issue is the device speed. The most important timing consideration for combinatorial PLDs is the propagation delay (t_{PD}) of signals from the input to the output of the device. For registered PLDs, the important timing consideration is the device clocking frequency. This clocking frequency is in turn determined by sum of the register setup time (t_S), and clock-to-output propagation delay (t_{CO}). Most systems impose some timing restrictions on the internal logic functions. These restrictions will determine the necessary t_{PD} (for combinatorial devices) or f_{MAX} (for registered devices).

In our design example, the PLD will primarily perform address decoding. The critical system timing constraint is determined by the read/write cycle time of the microprocessor and the memory access time available (Figure 5). Most microprocessors allow anywhere from 10 to 35 ns for address decoding. That is, 10 ns – 35 ns after the address is available, the correct memory chip-select signal should be asserted. In our design example, the available cycle time of 240 ns and memory access time of 220 ns leaves barely 10 ns for address decode time. We can check the propagation delay and select the appropriate speed device for our design, which is t_{PD} = 10 ns.

We have already briefly discussed the types of applications where registers are needed. Sometimes the consideration of system timing can affect whether or not registers are needed. Devices with registers can hold a signal stable for the long durations required by the addressed peripheral or memory. However, this slows the initial response or access time of the device since the chip select must wait for the setup time before the rising edge of the clock cycle. Devices without registers provide fast access time but hold the signal valid only as long as the input conditions are valid. In most address decoders, the address signals are kept asserted by the microprocessor until the read/write cycle is completed. In this case, the registers are not required for holding the signals asserted.

The remaining two general design considerations are the number of product terms and output polarity. We will discuss these two as we implement the design in the next section.

Implementing a Design

Implementing a design (Figure 6) requires the creation of a design file. The design file contains three types of information.

- Basic bookkeeping information
- Design syntax
- Simulation syntax

Once the design file is complete, it is then assembled and simulated. Once it passes assembly and simulation, the resultant JEDEC file is downloaded to a device programmer for configuring the device.

Design Syntax

In this example, as shown in Figure 6, there are two options available to the designer for expressing the design. The first is through traditional Boolean logic equations; the second is through a state machine syntax. The Boolean logic equations are the only option for combinatorial designs and can also be efficient for some registered designs. The Boolean equations can be derived from a combination of the functional description, the truth table and/or the timing diagrams (Figure 7). The state machine approach is ideal for large registered control designs, and can be derived from the functional description, state table, state diagram and/or the timing diagram (Figure 8).

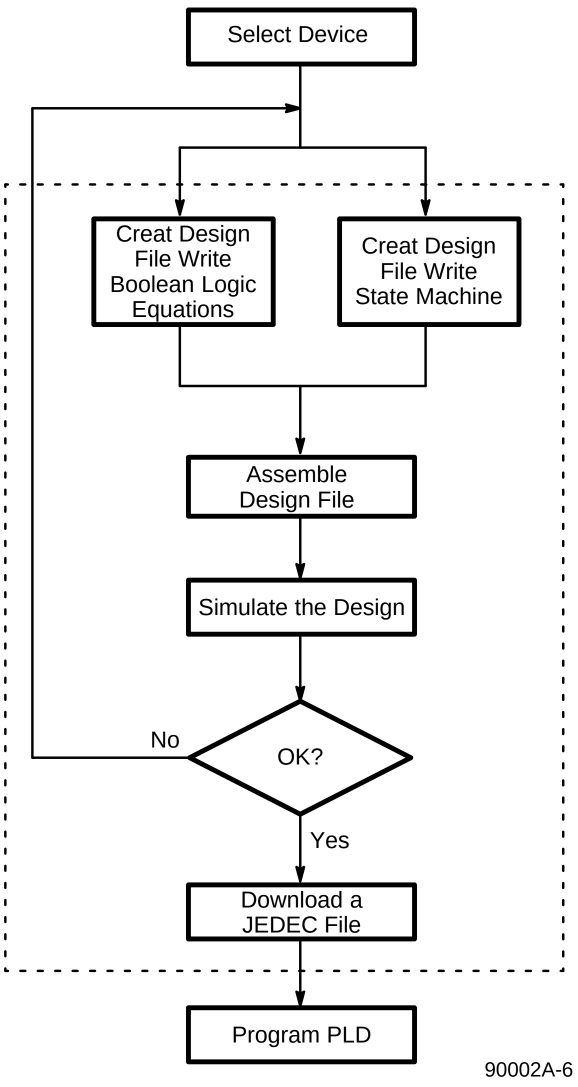


Figure 6. Implementing a Design

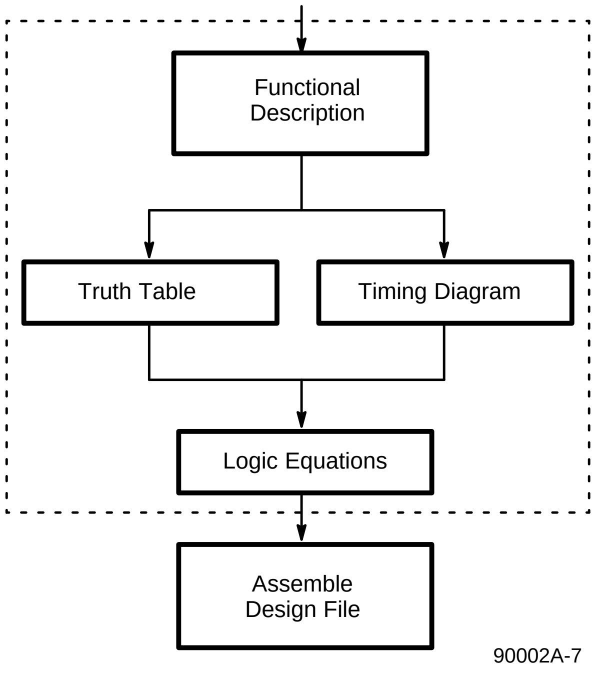


Figure 7. Writing Boolean Logic Equations

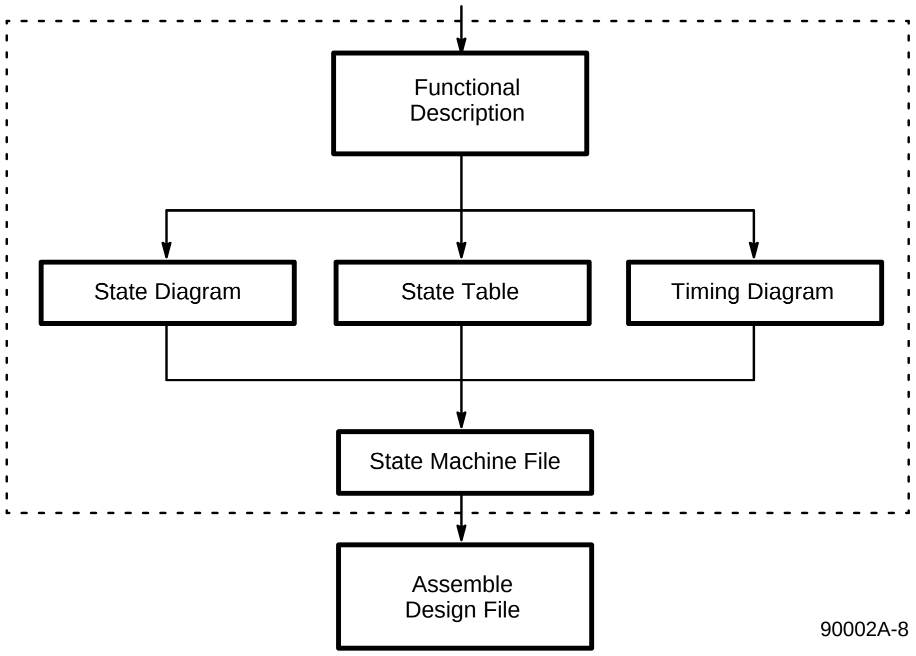


Figure 8. State Machine Description

Boolean Logic Equations

Boolean equations are used to represent the sum-of-products logic form. The Boolean equations are ideally suited for representing the two-level AND-OR logic available in most PLDs.

A conventional approach to the design is to convert the design problem to its discrete logic implementation. Such random SSI and MSI logic can be easily implemented in PLDs. This usually involves converting to sum-of-products Boolean logic form. This approach can be a chore, and much effort can be saved by implementing a design with PLDs in a sum-of-products form right from the start. This essentially means that the designer does not have to design around the limitations of fixed SSI and MSI functions. A direct implementation of a design in sum-of-products form in a PLD can also yield a faster circuit.

Boolean equations can be directly derived from the truth table or timing diagram (Figure 7). The truth table is used more often in simple combinatorial designs. The timing diagram method is used more often in registered control designs. We will first discuss the truth table method and then discuss the details of the timing diagram method.

In addition to specifying the logic function, the Boolean equations in the design file help document the design. There is no need to draw out an equivalent schematic. This allows design modularity; the schematic can just show a block for a particular PLD. Separate supporting documentation (the design file) provides the details without cluttering the drawing.

Truth-Table-Based Design

The requirements for our particular design example can be easily converted to a truth table format (Table 2). This truth table is based upon the functional description of the design, and is derived from the address map (Figure 3) and the truth table (Table 1).

Table 2. Truth Table for the Address Decoder

A23	A22	A21	INIT	AS	RW	Output Generated		
						ROMCS1	ROMCS2	RAMCS
0	0	0	1	0	1	0	1	1
0	0	1	1	0	1	1	0	1
0	1	0	1	0	X	1	1	0
0	1	1	1	0	X	1	1	0
1	0	0	1	0	X	1	1	0
1	0	1	1	0	X	1	1	0

There are three additional input signals in this design example. The first, RW, is generated by the microprocessor, and distinguishes between read and write cycles. Since the ROM data is only for reading, the ROMCS1 and ROMCS2 signals are asserted only when RW is high (when the microprocessor attempts to read the ROM) and are not asserted for the write cycle. On the other hand, RAMCS is generated for both read and write cycles and the state of signal RW is “don’t care.”

The second additional signal, AS, is the address strobe signal generated by the microprocessor, and is asserted only when the address lines carry a valid address. All of the chip select signals need to be gated with the AS signal to ensure that they are only generated for valid addresses, and no spurious chip selects are generated.

The last signal is the INIT signal, which is a system initialization signal. This signal is used to initialize the microprocessor for a “warm boot,” and none of the chip selects is allowed when this INIT signal is asserted.

Writing Boolean equations from the above logic is very straight forward. The output signal names, along with their polarity, are assigned to sum-of-product equations, which are based upon inputs and their polarities.

```
/ROMCS1 = /A23 * /A22 * /A21 * INIT * /AS * RW
/ROMCS2 = /A23 * /A22 *  A21 * INIT * /AS * RW
/RAMCS  = /A23 *  A22 * /A21 * INIT * /AS
        + /A23 *  A22 *  A21 * INIT * /AS
        +  A23 * /A22 * /A21 * INIT * /AS
        +  A23 * /A22 *  A21 * INIT * /AS
```

Figure 9. The Implementation In Boolean Equations

The equations are derived directly from the truth tables. Each one of the AND equations uses up one product term of the device as shown in Figure 9. One device selection consideration is to ensure that all the outputs have sufficient product terms to accommodate the desired function.

This brings us to the issue of output polarity. Suppose we had to generate active-HIGH outputs. In that case the output equations for the ROMCS1 signal would be:

```
ROMCS1 = /A23 + /A22 * /A21 * INIT */AS * RW
```

If the device has active-LOW outputs only, this equation’s output polarity needs to be inverted to be able to fit the device. Using DeMorgan’s theorem for Boolean logic we get:

```
/RCMCS1 = A23 + A22 + A21 + /INIT + AS + /RW
```

This equation requires a large number of product terms (six). Some signals are efficient and use fewer product terms in their true form, while others are more efficient in their inverted form. The device selection issues of product terms and output polarity also apply to registered designs.

Timing-Diagram-Based Design

Until now, we have discussed a PLD design using truth tables as the primary design vehicle. In this section we will attempt a design using a timing diagram as a design vehicle.

Earlier in the address decoder design we mentioned the INIT signal. This INIT signal essentially an initialization signal for the entire system. The INIT signal is used internally (via feedback) for disabling the chip selects during initialization. Externally it can be used to initialize other system signals. This INIT signal is generated from a RESET switch connected to the inputs of the device as shown in Figure 10.

Most experienced designers understand the tradeoffs for device selection. They implicitly go through the steps of design conceptualization and device selection, explained earlier. They typically draw a block around the logic being designed, with the previous knowledge that it would fit a PLD which has sufficient inputs, outputs, IOs and product terms.

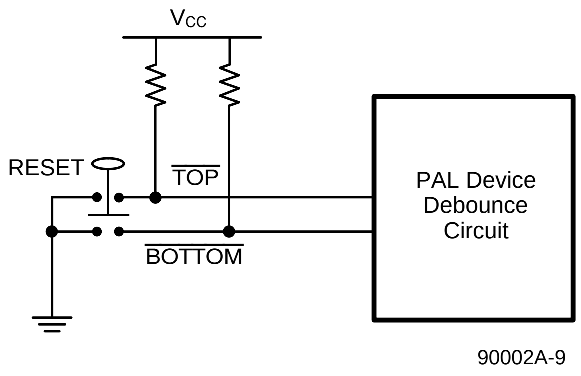


Figure 10. RESET Switch for System Initialization

To avoid unwanted initialization, the RESET switch must be debounced. That is, we want the INIT signal to remain HIGH until the switch actually contacts the bottom side. Once the bottom side is hit, INIT should be asserted active LOW. Once asserted, it should stay LOW and not change until the top side is hit again. The timing requirements of the debounce circuitry are shown in Figure 11. Signals TOP and BOTTOM are inputs to the programmable logic device. These signals are activated when the RESET switch touches the top and the bottom contacts, respectively.

We can formulate the equations by looking at the timing requirements of the debounce circuitry shown in Figure 11. The idea is to identify the key elements of this timing diagram. The arrows in Figure 11 show the critical events. The first arrow shows the normal state of all the pins when the RESET switch is not asserted. Subsequent arrows show each event in the timing of the INIT signal, depending upon the movement of the switch.

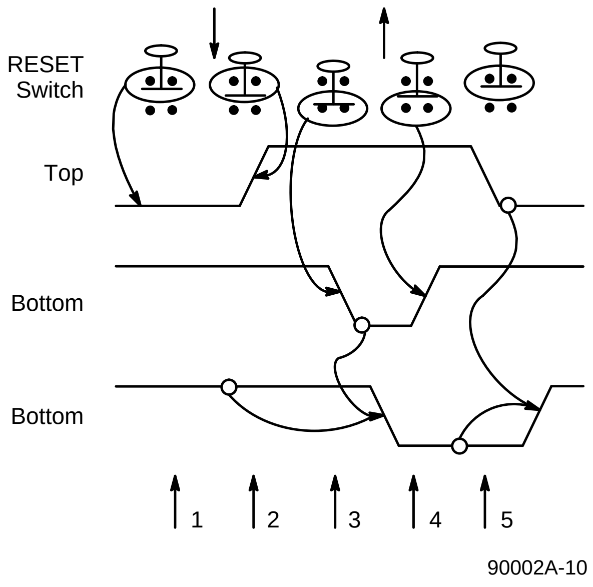


Figure 11. Timing Diagram for the Debounce Switch

The logic level of the signals at each critical event carries useful logic information for deriving Boolean equations. This logic information for each event is converted into direct Boolean equations as shown in below. For example, at instant 1 the INIT signal remains HIGH as long as the TOP signal remains LOW; this is converted to INIT = /TOP * BOTTOM.

1. Normal state	`INIT = /TOP`
2. Switch travels from TOP to BOTTOM	`INIT = TOP * BOTTOM * INIT`
3. Switch contacts BOTTOM	`/INIT = /BOTTOM`
4. Switch travels from BOTTOM to TOP	`/INIT = /INIT * BOTTOM * TOP`
5. Normal State Again	

We can combine the two active-LOW events into one equation:

```
/INIT = /BOTTOM
      + /INIT * BOTTOM * TOP
```

Minimizing, this becomes:

```
/INIT = /BOTTOM
      + / INIT * TOP
```

This can also be done by way of a truth table and Karnaugh map.

Table 3. Truth Table of INIT Logic

TOP	BOTTOM	INIT–	INIT+
1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	1
0	0	1	X
0	0	0	X

Here TOP or BOTTOM will be LOW if contacted. Note that both TOP and BOTTOM can not be contacted at the same time. The truth table of Table 3 yields the Karnaugh map shown in Figure 12. Grouping the zeros (because we are using active-LOW outputs) yields the Boolean equation identical to the one derived from the timing diagram.

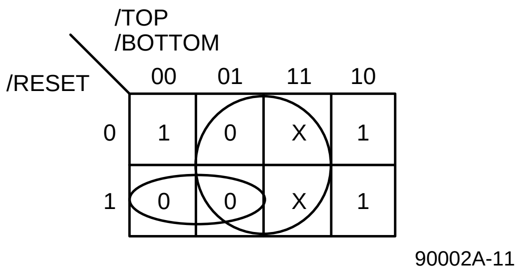


Figure 12. Karnaugh Map of INIT Signal Logic

There is essentially no difference between the truth table and timing diagram techniques for writing Boolean logic. Also, a careful analysis will indicate that we implicitly assumed a truth table in the timing diagram example. Some designers prefer to make a separate truth table (at least in the first few PLD designs), while others prefer to design directly from timing diagrams. While the truth table method allows a more optimal utilization of product terms, the timing diagram method is easier to visualize as it retains the design perspective. In both cases the logic should be minimized by the design software to ensure that the design is testable.

Most experienced designers understand the tradeoffs for device selection. They implicitly go through the steps of design conceptualization and device selection, explained earlier. They typically draw a block around the logic being designed, with the previous knowledge that it would fit a PLD which has sufficient inputs, outputs, IOs and product terms.

Simulation

Design simulation is an integral part of the design process, as shown in Figure 13. The purpose is to exercise all of the inputs and test the response of outputs to verify that they will work as desired in the system. These are essentially test vectors which designate the state of every input on the device; the outputs are then checked for an appropriate response. The simulation test vectors identify any flaws in the design equations which could affect the logical operation of the devices programmed. Thus, the simulation vectors serve as a design debugging tool.

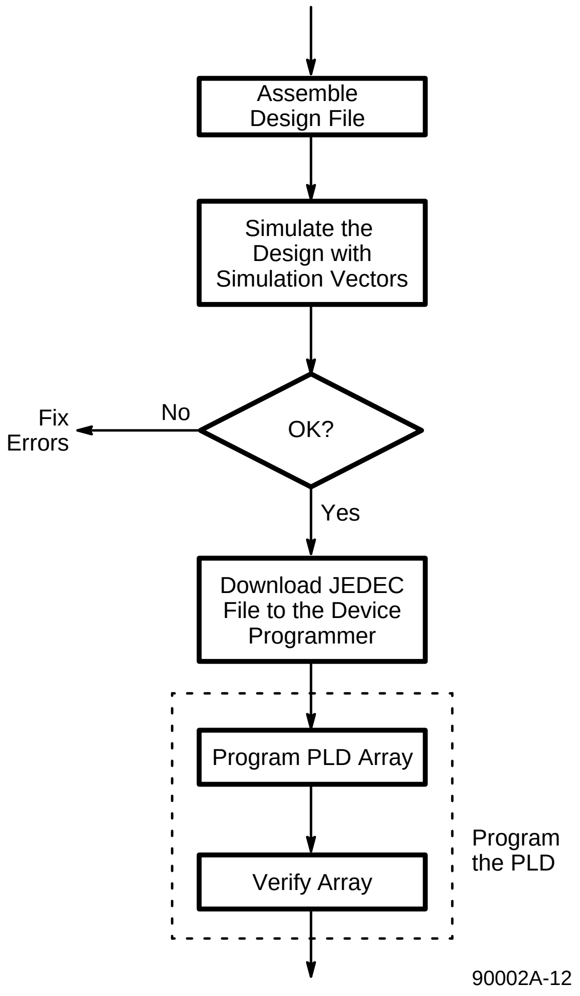


Figure 13. Device Simulation and Programming

Simulation test vectors will eventually make up part of a larger set of test vectors called ”functional test vectors”. These functional test vectors are used to exercise a real device after programming to identify any individual devices which are defective. Other means of identifying defective devices, such as signature analysis, are also available. In this section we will strictly focus on simulation vectors.

Simulation is included in the design file along with the logic equations. There is little standardization in these simulation expressions among various PLD design software packages, although most of them rely on test vectors to exercise the logic.

The simulation vectors or events can be directly derived from the truth table and the timing diagram of the design. The logic level and functions of all signals can be expanded and rewritten in a test vector form by the software. For example, the truth table for the address decoder example discussed earlier can be easily rewritten as shown in Table 4.

Table 4. Truth Table Used to Derive Simulation Vectors

A23	A22	A21	TOP	BOTTOM	AS	RW	ROMCS1	ROMCS2	RAMCS	INIT
0	0	0	0	1	1	1	H	H	H	H
0	0	0	0	1	0	1	L	H	H	H
0	0	1	0	1	1	1	H	H	H	H
0	0	1	0	1	0	1	H	L	H	H
0	1	0	0	1	1	X	H	H	H	H
0	1	0	0	1	0	X	H	H	L	H
0	1	1	0	1	1	X	H	H	H	H
0	1	1	0	1	0	X	H	H	L	H
1	0	0	0	1	1	X	H	H	H	H
1	0	0	0	1	0	X	H	H	L	H
1	0	1	0	1	1	X	H	H	H	H
1	0	1	0	1	0	X	H	H	L	H
1	0	1	0	1	X	X	H	H	H	H
1	0	1	1	1	X	X	H	H	H	H
1	0	1	1	0	1	X	H	H	H	L
1	0	1	1	1	1	X	H	H	H	L
1	0	1	0	1	1	X	H	H	H	H

These are essentially the simulation vectors which will allow us to define the inputs to the device and check the outputs of the device.

The simulator then interprets the design file and generates the output logic levels and/or waveforms, which can be checked by the designer.

Once the simulation is complete, the design file can be assembled to generate the JEDEC file. In the proceeding discussions we have assumed prior knowledge of the design file assembly. The procedure for assembly varies with different software packages.

Device Programming and Testing

Once the design simulation is completed, the final step is device programming and testing (Figure 14). Programmers are available from a variety of vendors. It is important to note that Advanced Micro Devices, Inc., qualifies programmers upon verifying that the algorithms used by the programmers are correct and that other basic criteria are met. When purchasing a programmer, check that the programmer is qualified for the devices you intend to use.

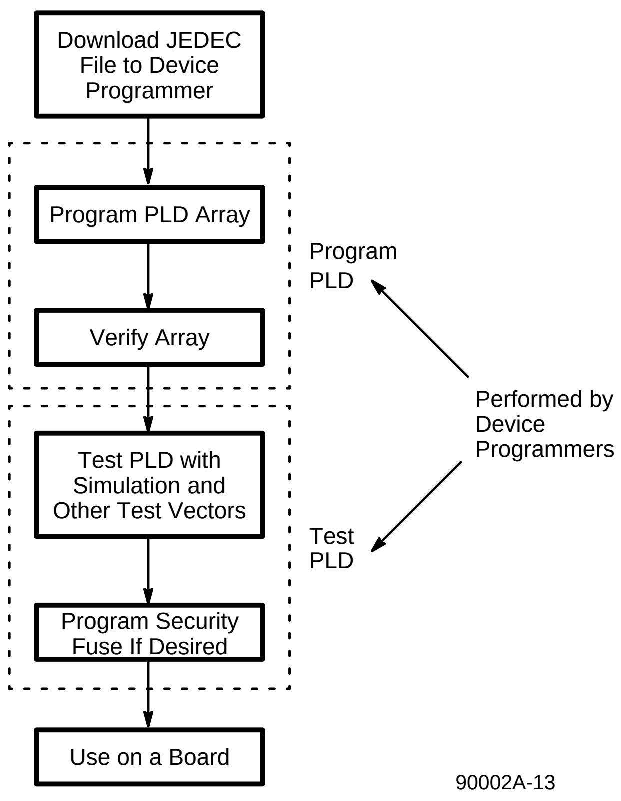

Download JEDEC File to Device Programmer
Program PLD Array
Verify Array
Test PLD with Simulation and Other Test Vectors
Program Security Fuse If Desired
Use on a Board
Program PLD
Performed by Device Programmers
Test PLD
90002A-13

Figure 14. Device Programming and Testing

There are two types of programmers available: menu-driven or device code based. The menu-driven programmer directly indicates the part type being programmed, whereas the latter type requires the user to enter the device code before programming.

Once the JEDEC fuse file has been downloaded, the programmer can program the device; the PLD is then ready for use. The programmer also verifies the connections after the programming cycle. Programmers also provide the capability of reading a previously programmed device and creating duplicates of that device.

Testing PLDs

The testing of PLDs can be performed by the device programmer or by other test equipment. For a manufacturing environment, where high yields are required, device testing is critical. After testing is complete, the device security bit may be programmed, if desired, to secure the design from copying.